Constructors and Destructors

6

Key Concepts

Constructing objects | Constructors | Constructor overloading | Default argument constructor | Copy constructor | Constructing matrix objects | Automatic initialization | Parameterized constructors | Default constructor | Dynamic initialization | Dynamic constructor| Destructors

6.1 Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata( )** and **setvalue( )** to provide initial values to the private member variables. For example, the following statement

```
A.input( );
```

invokes the member function **input( ),** which assigns the initial values to the data items of object **A.** Similarly, the statement

```
x.getdata(100,2 9 9.95);
```

passes the initial values as arguments to the function **getdata( ),** where these values are assigned to the private variables of object **x.**All these ‘function call’ statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class,**that behave very similar to the built-in types. This means that we should be able to initialize a **class**type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization*of objects. It also provides another member function called the *destructor*that destroys the objects when they are no longer required.

6.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor
class integer
{
int m, n;
public:
```

```
integer(void); // constructor declared
. . . . .
. . . . .
};
integer :: integer(void) // constructor defined
{
m = 0; n = 0;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1; // object intl created
```

not only creates the object **intl**of type **integer**but also initializes its data members **m**and **n**to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor.*The default constructor for **class A is A::A( ).**If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object **a.**

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.

- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual.**(Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make ‘implicit calls’ to the operators **new**and **delete**when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

6.3 Parameterized Constructors

The constructor **integer( ),**defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors.*

The constructor **integer( )**may be modified to take arguments as shown below:

```
class integer {
int m, n;
public:
integer(int x, int y); // parameterized constructor
. . . . .
. . . . .
};
integer :: integer(int x, int y)
{
m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100); // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

Program 6.1 Class with Constructors

```
#include <iostream>
using namespace std;
class integer
{
int m, n;
public:
integer(int, int); // constructor declared

void display(void)
{
cout << “ m = “ << m << “\n”
cout << “ n = “ << n << “\n”
}
};
integer :: integer(int x, int y) // constructor defined
{
m = x; n = y;
}
int main( )
{
integer int1(0,100); // constructor called implicitly
integer int2 = integer(25, 75); // constructor called explicitly
cout << “\nOBJECT1“ << “\n”
int1.display( );
cout << “\nOBJECT2“ << “\n”
int2.display( );
return 0;
}
```

The output of program 6.1 would be:

```
OBJECT1
m = 0 n = 100

OBJECT2
```

```
m = 25
n = 75
```

The constructor functions can also be defined as **inline**functions. Example:

```
class integer
{
int m, n;
public:
integer(int x, int y)  // Inline constructor
{
m = x; y = n;
}
. . . . .
. . . . .
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
. . . . .
. . . . .
public:
A(A);
};
```

is illegal.

However, a constructor can accept a *reference*to its own class as a parameter. Thus, the statement

```
Class A
{
. . . . .
. . . . .
public:
A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor.*

6.4 Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer( ); // No arguments
integer(int, int); // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main( ).**C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
int m, n;
public:
integer( ){m=0; n=0;} // constructor 1
integer(int a, int b)
{m = a; n = b;} // constructor 2
integer(integer & i)
{m = i.m; n = i.n;} // constructor 3
};
```

This declares three constructors for an **integer**object. The first constructor receives no arguments, the second receives two **integer**arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m**and **n**of **I1**to zero. The statement

```
integer I2(20,40);
```

would call the second constructor which will initialize the data members **m**and **n**of **I2**to 20 and 40 respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2**into **I3.**In other words, it sets the value of every data element of **I3**to the value of the corresponding data element of **I2.**As mentioned earlier, such a constructor is called the *copy constructor.*We learned in Chapter 4 that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program 6.2 shows the use of overloaded constructors.

Program 6.2 Overloaded Constructors

```
#include <iostream>
using namespace std;
class complex
{
float x, y;
public:
complex( ){ }  // constructor no arg
complex(float a) {x = y = a;}  // constructor-one arg
complex(float real, float imag)  // constructor-two args
{x = real; y = imag;}
friend complex sum(complex, complex);
friend void show(complex);
};
complex sum(complex c1, complex c2)  // friend
{
complex c3;
c3.x = cl.x + c2.x;
c3.y = cl.y + c2.y;
return(c3);
}
void show(complex c)  // friend
{
cout << c.x << " + j" << c.y << "\n"
}
int main( )
{
complex A(2.7, 3.5);  // define& initialize
complex B(1.6);  // define& initialize
complex C;  // define
C = sum(A, B);  // sum( ) is a friend
cout << "A = "; show(A);  // show( ) is also friend
cout << "B = "; show(B);
cout << "C = "; show(C);
// Another way to give initial values(second method)
complex P,Q,R; // define P, Q and R
P = complex(2.5,3.9); // initialize P
Q = complex(1.6,2.5); // initialize Q
R = sum(P,Q);
cout << "\n"
cout << "P = "; show(P);
cout << "Q = "; show(Q);
cout << "R = "; show(R);
return 0;
}
```

The output of Program 6.2 would be:

```
A = 2.7 + j3.5
```

```
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

NOTE: *There are three constructors in the class* ***complex.****The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.*

Let us look at the first constructor again.

```
complex( ){ }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? As pointed out earlier, C++ compiler has an *implicit constructor*which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the “do-nothing” implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

6.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex( ) can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag**is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real**variable and 0.0 to **imag**(by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real**and 3.0 to **imag.**The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A( )**and the default argument constructor **A::A(int = 0).**The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A( ) or A::A(int = 0).**

6.6 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 6.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

Program 6.3 Dynamic Initialization of Objects

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
long int P_amount;  // Principal amount
int Years;  // Period of investment
float Rate;  // Interest rate
float R_value;  // Return value of amount
public:
Fixed_deposit( ){ }
Fixed_deposit(long int p, int y, float r=0.12);
Fixed_deposit(long int p, int y, int r);
void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
P_amount = p;
Years = y;
Rate = r;
R_value = P_amount;
for(int i = 1; i <= y; i++)
```

```
R_value = R_value * (1.0 + r);
Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
P_amount = p; Years = y; Rate = r;
R_value = P_amount; for(int i=1; i<=y; i++)
R_value = R_value*(1.0+float(r)/100);
}
void Fixed_deposit :: display(void)
{
cout << “\n”
   << “Principal Amount = “ << P_amount << “\n”
   << “Return Value = “ << R_value << “\n”
}
int main( )
{
Fixed_deposit FD1, FD2, FD3;  // deposits created
long int p;  // principal amount
int y;  // investment period, years
float r;  // interest rate, decimal form
int R;  // interest rate, percent form
cout << “Enter amount, period,interest rate(in percent)”<<“\n”
cin >> p >> y >> R;
FD1 = Fixed_deposit(p,y,R);
cout << “Enter amount, period, interest rate(decimal form)” <<
“\n”
cin >> p >> y >> r;
FD2 = Fixed_deposit(p,y,r);
cout << “Enter amount and period” << “\n”
cin >> p >> y;
FD3 = Fixed_deposit(p,y);
cout << “\nDeposit 1” FD1.display( );
cout << “\nDeposit 2” FD2.display( );
cout << “\nDeposit 3” FD3.display( );
return 0;
}
```

```
The output of Program 6.3 would be:
Enter amount,period,interest rate(in percent) 10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
Return Value = 16430.3

Deposit 2
Principal Amount = 10000
Return Value = 16430.3

Deposit 3
Principal Amount = 10000
Return Value = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

NOTE: *Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for**r.***

6.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of **I1.**Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization.*Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1**and **I2**are objects, this statement is legal and simply assigns the values of **I1**to **I2,**member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 6.4.

Program 6.4 Copy Constructor

```
#include <iostream>
using namespace std;
class code
{
int id;
public:
code( ){ }  // constructor
code(int a) { id = a;}  // constructor again
code(code & x)  // copy constructor
{
id = x.id;  // copy in the value
}
void display(void)
{
cout << id;
}
};
int main( )
{
code A(100);  // object A is created and initialized
code B(A);  // copy constructor called
code C = A;  // copy constructor called again
code D;  // D is created, not initialized
D = A;  // copy constructor not called
cout << "\n id of A: "; A.display( );
cout << "\n id of B: "; B.display( );
cout << "\n id of C: "; C.display( );
cout << "\n id of D: "; D.display( );
return 0;
}
```

The output of Program 6.4 would be:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

NOTE:*A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.*

When no copy constructor is defined, the compiler supplies its own copy constructor.

6.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 6.5 shows the use of new, in constructors that are used to construct strings in objects.

Program 6.5 Constructors with new

```
#include <iostream>
#include <string>
using namespace std;
class String
{
char *name;
int length;
public:
String( )  // constructor-1
{
    length = 0;
    name = new char[length + 1];
    }
    String(char *s)  // constructor-2
    {
    length = strlen(s);
    name = new char[length + 1];  // one additional // character
for\0
    strcpy(name, s);
    void display(void)
    {cout << name << “\n”}
    void join(String &a, String &b);
    };
    void String :: join(String &a, String &b)
    {
    length = a.length + b.length; delete name;
    name = new char[length+1]; // dynamic allocation
    strcpy(name, a.name); strcat(name, b.name);
    };
    int main( )
    {
    char *first = “Joseph “;
    String name1(first), name2(“Louis “), name3 (“Lagrange”),
    s1,s2;

    s1.join(name1, name2);
    s2.join(s1, name3);
    name1.display( );
    name2.display( );
    name3.display( );
    s1.display( );
    s2.display( );

    return 0;
    }
```

The output of Program 6.5 would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

NOTE: *This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the* ***length*** *of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.*

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a.** The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

6.9 Constructing Two-Dimensional Arrays

We can construct matrix variables using the class type objects. The example in Program 6.6 illustrates how to construct a matrix of size m x n.

Program 6.6 Constructing Matrix Objects

```
#include <iostream>
using namespace std;
class matrix
{
int **p; // pointer to matrix
int d1,d2;  // dimensions
public:
matrix(int x, int y);
void get_element(int i, int j, int value)
{p[i][j]=value;}
int & put_element(int i, int j)
{return p[i][j];}
};
matrix :: matrix(int x, int y)
{
d1 = x; d2 = y;
p = new int *[d1];  // creates an array pointer
for(int i = 0; i < d1; i++)
p[i] = new int[d2];  // creates space for each row
}
int main( )
{
int m, n;

cout << “Enter size of matrix: “
cin >> m >> n;
matrix A(m,n);  // matrix object A constructed

cout << “Enter matrix elements row by row \n”
int i, j, value;

for(i = 0; i < m; i++)
for(j = 0; j < n; j++)
```

```
    {
    cin >> value;
    A.get_element(i,j,value);
    }
    cout << "\n"
    cout << A.put_element(1,2);
    return 0;
    };
```

The output of Program 6.6 would be:

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22
17
```

17 is the value of the element (1, 2).

The constructor first creates a vector pointer to an **int**of size **d1.**Then, it allocates, iteratively an **int**type vector of size **d2**pointed at by each element **p[i].**Thus, space for the elements of a d1 x d2 matrix is allocated from free store as shown above.

6.10 const Objects

We may create and use constant objects using **const**keyword before object declaration. For example, we may create X as a constant object of the class **matrix**as follows:

```
const matrix X(m,n);  // object X is constant
```

Any attempt to modify the values of **m**and **n**will generate compile-time error. Further, a constant object can call only **const**member functions. As we know, a **const**member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const**objects try to invoke **nonconst**member functions, the compiler generates errors.

6.11 Destructors

A *destructor,*as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer( ){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new**is used to allocate memory in the constructors, we should use **delete**to free that memory. For example, the destructor for the **matrix**class discussed above may be defined as follows:

```
matrix :: ~matrix( )
{
for(int i=0; i<d1;
delete p[i];
delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

Program 6.7 Implementation of Destructors

```
#include<iostream>
using namespace std;
int count=0;
class test
{
public:
test( )
{
count++;
cout<<"\n\nConstructor Msg: Object number "<<count<<
"created.."
}
~test( ) {
cout<<"\n\nDestructor Msg: Object number "<<count<<"
destroyed.."
count--;
}
};
int main( )
{
cout<<"Inside the main block.." cout<<"\n\nCreating first object
T1.."
test T1;
{ //Block 1
cout<<"\n\nInside Block 1.."
cout<<"\n\nCreating two more objects T2 and T3.."
test T2,T3;
cout<<"\n\nLeaving Block 1.."
}
cout<<"\n\nBack inside the main block.."
return 0;
}
```

The output of Program 6.7 would be:

```
Inside the main block..
Creating first object T1..
Constructor Msg: Object number 1 created..
Inside Block 1..
Creating two more objects T2 and T3..
Constructor Msg: Object number 2 created..
Constructor Msg: Object number 3 created..
Leaving Block 1..
Destructor Msg: Object number 3 destroyed.
Destructor Msg: Object number 2 destroyed.
Back inside the main block..
Destructor Msg: Object number 1 destroyed..
```

NOTE:*A class constructor is called everytime an object is created. Similarly, as the program control leaves the current block the objects in the block start getting destroyed and destructors are called for each one of them. Note that the objects are destroyed in the reverse order of their creation. Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside main.*

Similar functionality as depicted in Program 6.7 can be attained by using static data members with constructors and destructors. We can declare a static integer variable count inside a class to keep a track of the number of its object instantiations. Being static, the variable will be initialized only once i.e., when the first object instance is created. During all subsequent object creations, the constructor will increment the count variable by one. Similarly, the destructor will decrement the count variable by one as and when an object gets destroyed. To realize this scenario, the code in Program 6.7 will change slightly, as shown below:

```
#include <iostream>
using namespace std;

class test
{
private:
static int count=0;
public:
.
.
}
test ( )
{
.
count++;
}
~test ( )
{
count-- ;
}
```

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. Program 6.8 demonstrates how a destructor releases the memory allocated to an object:

Program 6.8 Memory Allocation to an Object Using Destructor

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
int *a;
public:
test(int size)
{
a = new int[size];
cout<<“\n\nConstructor Msg: Integer array of size “<<size<<“
created..”
}
~test( )
{
delete a;
cout<<“\n\nDestructor Msg: Freed up the memory allocated for
integer array”
}
};
int main( )
{
int s;
cout<<“Enter the size of the array..”
cin>>s;
cout<<“\n\nCreating an object of test class..”
test T(s);
cout<<“\n\nPress any key to end the program..”
getch( );

return 0;
}
```

The output of Program 6.8 would be:

```
Enter the size of the array..5
Creating an object of test class..
Constructor Msg: Integer array of size 5 created..
Press any key to end the program..
Destructor Msg: Freed up the memory allocated for integer array
```

Summary

- ❑ C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization*of objects.
- ❑ A constructor has the same name as that of a class.
- ❑ Constructors are normally used to initialize variables and to allocate memory.
- ❑ Similar to normal functions, constructors may be overloaded.
- ❑ When an object is created and initialized at the same time, a copy constructor gets called.
- ❑ We may make an object **const**if it does not modify any of its data values.
- ❑ C++ also provides another member function called the destructor that destroys the objects when they are no longer required.

Key Terms

automatic initialization | **Const** | Constructor | constructor overloading | copy constructor | copy initialization | default argument | default constructor | **Delete** | Destructor | dynamic construction | dynamic initialization | explicit call | implicit call | implicit constructor | initialization | **new** | parameterized constructor | reference | shorthand method | **strcat( )** | **strcpy( )** | **strlen( )** | **virtual**

Review Questions

6.1 What is a constructor? Is it mandatory to use constructors in a class?

6.2 How do we invoke a constructor function?

6.3 List some of the special properties of the constructor functions.

6.4 What is a parameterized constructor?

6.5 Can we have more than one constructors in a class? If yes, explain the need for such a situation.

6.6 What do you mean by dynamic initialization of objects? Why do we need to do this?

6.7 How is dynamic initialization of objects achieved?

6.8 Distinguish between the following two statements:

```
time T2(T1);
time T2 = T1;
```

T1 and T2 are objects of **time**class.

6.9 Describe the importance of destructors.

6.10 State whether the following statements are TRUE or FALSE.

(a) Constructors, like other member functions, can be declared anywhere in the class

(b) Constructors do not return any values.

(c) A constructor that accepts no parameter is known as the default constructor.

(d) A class should have at least one constructor.

(e) Destructors never take any argument.

(f) The objects are always destroyed in the reverse order of their creation.

Debugging Exercises

6.1 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
int length;
int width;
public:
Room(int l, int w=0):
width(w),
length(l)
{
}
};
void main( )
{
Room objRoom1;
Room objRoom2(12, 8);
}
```

6.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
int length;
int width;
public:
Room( )
{
length = 0;
width = 0;
{
Room(int value=8)
length = width = 8;
void display( )
cout << length << ‘ ‘ << width;
}
};
void main( )
{
Room objRoom1;
objRoom1.display( );
}
```

6.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
int width;
int height;
static int copyConsCount;
public:
void Room( )
{
width = 12;
height = 8;
}
Room(Room& r)
{
width = r.width;
height = r.height;
copyConsCount++;
}
void dispCopyConsCount( )
{
cout << copyConsCount;
}
};
int Room::copyConsCount = 0; void main( )
{
Room objRoom1;
Room objRoom2(objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;

objRoom4.dispCopyConsCount( );
}
```

6.4 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
int width;
int height;
static int copyConsCount;
public:
Room( ) {
width = 12;
height = 8;
}
Room(Room& r) {
width = r.width;
height = r. height;
copyConsCount++;
}
void disCopyConsCount( )
{
cout << copyConsCount;
}
};
int Room::copyConsCount = 0;
void main( )
{
Room objRoom1;
Room objRoom2 (objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;
objRoom4.dispCopyConsCount( );
}
```

Programming Exercises

6.1 Design constructors for the classes designed in Programming Exercises 5.1 through 5.5 of Chapter 5.

6.2 Define a class **String**that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string

```
String s1; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
String s2(“Well done!”);
```

Include a function that adds two strings to make a third string. Note that the statement

```
s2 = s1;
```

will be perfectly reasonable expression to copy one string to another.

Write a complete program to test your class to see that it does the following tasks:

(a) Creates uninitialized string objects.

(b) Creates objects with string constants.

(c) Concatenates two strings properly.

(d) Displays a desired string object.

6.3 A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.

Design a system using a class called **books**with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

6.4 Improve the system design in Exercise 6.3 to incorporate the following features:

(a) The price of the books should be updated as and when required. Use a private member function to implement this.

(b) The stock value of each book should be automatically updated as soon as a transaction is completed.

(c) The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use **static**data members to keep count of transactions.

6.5 Modify the program of Exercise 6.4 to demonstrate the use of pointers to access the members.